MPI BASICS

George VandenBerghe
IBM
June 20, 2005

Topics

- Introduction
- simple example
- Point to point messages
- forground v.s background (or blocking v.s. nonblocking)
- collectives (broadcasts, gathers, reductions, barriers, exchanges)
- Environment settings
- performance
- debugging

Other materials

- Presentation on ibmdocs/userman/mpiuse.pdf
- IBM Practical MPI Programming redbook (google ibm redbook mpi programming for a link)
- MPI Programming and Subroutine Reference (on ibmdocs, use for binding reference)
- Using MPI Gropp, Lusk and Skjellum 1999 MIT Press (should be at Amazon)

Examples

- An example of a reduce is in /nfsuser/g02/courses/mpi/reduce.f
- An example of a scatter, transpose, gather is in /nfsuser/g02/courses/mpi/coll.f
- Will work on a simple halo exchange but not ready for 6/20. This will be in laplace.f
- A sample LL job is in /nfsuser/g02/courses/mpi/j

Why MPI?

- Size and cost of algorithms has grown even faster than cpu speed.
- We need to use several cpus in parallel to get our speeds (several can be ~1000).
- One paradigm is for one program to spawn additional threads and break up array operations or other large chunks of work among threads.
- Standard for this is OPEN_MP.
- With OPEN_MP you run one program and it sends work to other cpus.
- This does not work over networks or switches (at least today)

Why MPI?

- With MPI you run N programs on N cpus and they communicate with each other through MESSAGES. This does work across networks (even WANs if comms are small and few). With current switch and cpu metrics scalability up to about 1000 tasks* is readily achieved for weather, physics, and CFD problems.
- Both OPEN_MP and MPI can be used in the same program. MPI tasks on N multicpu nodes are sped up by making OPEN_MP parallel regions.

What is MPI

MPI is a message passing API (applications programming interface)
It has become the standard and all major vendors support it.
MPI jobs consist of numerous executing programs (or copies of the same program) which execute concurrently and communicate with each other at intervals.
These are called TASKS
Information from other tasks is used as data for a given task or to control its flow.
Communication is done through calls to communication routines in the MPI library.
Communication is very visible in your program's source. Forced visibility has the advantage of focusing attention on a code's major cost points.

What is MPI

At low levels there are only a few primitive operations.
"send $this_much to $somewhere"
receive $this_much from $somewhere.

The other MPI calls are built on top of these or provide diagnostic information.

Most users work at a higher level of abstraction.
(Add this array across all tasks, exchange this with all tasks, collect all tasks arrays to this task, synchronize tasks,)

HARDWARE SUPPORT

- MPI can run on heterogeneous networks of machines of different types but in practice this is extremely rare.
- All use at NCEP is on clusters of individual machines of the same type connected by either a high speed communications network (the IBM supercomputers) or some kind of bus or crossbar, (some linux clusters, SGI) or sharing memory (IBM,SGI,CRAY, others)

RESOURCE USE

- When you run an MPI program as N tasks, you have the N cpus for the time the program is running.
- For efficiency they all need to have about the same amount of work to do.
- Departure from this is called LOAD IMBALANCE. Fast tasks wait for slow ones to catch up or finish.
- This is a major impediment to both scalability and dynamic scheduling. MPI programs must not time slice unless the whole gang time slices together.. only recently supported and still difficult to tune.
- Charging algorithms often charge for the time you have a job slot whether the cpu is active or not. No one else can safely use it while your job is active.

HARDWARE SUPPORT

- NCEP programming paradigms assume point to point speeds >100mb/sec, and comms rates independent of other activity on the fabric (how fast you go doesn't depend on what someone else is communicating). The MPI libraries were also written for that kind of platform.
- On the IBM clusters, two tasks on the same node communicate through memory. Tasks on different nodes communicate through the communications fabric or ***switch***

HOW FAST?

Point to point stream rates varied from 130mb/sec on asp/bsp in 2001, to 4gb/sec. on blue/white.

Small message latency declined from 25us to 10us over this time.

This compares with single stream memory bandwidth of 1000mb/sec and latency of 0.2us to memory.

When communicating we stream at appox. memory rates but latency is 50x or more higher (usually much higher)

HOW FAST?

Weather models are mostly bandwidth constrained.
Bandwidth even on the old systems was sufficient. Today
we have plenty and it's not an issue. These metrics
may vary on other vendors offerings. HPC industry experience has been
that customers like latency but buy bandwidth.

The two basic methods of PE integration are finite differences
and spectral transform.. term evaluation.. inverse transform.

Both scale to ~1000 tasks with our switch metrics. This
is also true for other vendors. The problem maps
well to this programming paradigm.

Scaling issues deferred to a later slide.

High level operations

Send arrays to and get arrays from other tasks (point to points)
Broadcast from one task to all of the others
Collect from all tasks to one.
Transpose problem domain.
Synchronize tasks.
Add (actually perform any reduction) across tasks.

After some experience you'll think in terms of these
high level operations rather than the code underneath.

Note I/O is outside the scope of MPI-1 (the most common
standard today). It is done by one or more tasks with user
algorithms. The quickest to understand is read on one
task.. broadcast or scatter to everyone; gather from everyone
then write from one.
Later (MPI-2) has a distributed I/O API but is outside scope of this class.

High level operations

For all MPI problems you break your problem into chunks that can run in parallel. This is a design or conversion issue. The process is called DECOMPOSITION

Finite difference problems operating on a 2D or 3D grid, break the grid into subsets and each task operates on a subset.

A special case of this is where one dimension is equal to the number of tasks and there are no comms along that direction. Then each processor gets a slice of the domain. In the more general case each processor gets more than one slice (few tasks) or has to further split slices (many tasks).

Quickly implemented codes hardwire the subgrid sizes and must run on a fixed number of tasks. Debugging is a little easier this way and it's not a bad way to get started.

Most production codes can run on varying number of tasks and the domain decomposition is done dynamically (GFS) , or with parameters included at compile time (NAM, george's primitive 1D RSM). This is usually worth spending the time to add in.

High level operations

The older GFS splits vertical levels between the tasks in spectral space. In cartesian space it breaks the horizontal domain into subdomains which have all levels but only a few latitudes and longitudes. The problem domain must be transposed (simple, presented later) to switch between these two decompositions.

The newer GFS decomposes by wavenumber and each task gets one or more wavenumbers.

Both the grid point partitioning and the GFS partitioning are exampled of DOMAIN DECOMPOSITION.

A second decomposition is FUNCTIONAL decomposition. Simple case is one processor does ocean and one processor does atmosphere in a coupled model (more generally many do the atmosphere and a few do the ocean part).

A second example is a model and post processor(s) running together with post processors getting forecast K from the model and processing it while the model timesteps to K+T where T is forecast interval. A subset of this sending all of the data to one processor which JUST does I/O in the background while the remaining processors integrate.

Numbering conventions

- MPI tasks start from task O, a four task job will spawn tasks 0,1,2,and 3.

A simple program

```
      include 'mpif.h'
      integer STATUS(mpi_status_size)
    integer itag
     call mpi_init(ier)
     a=0
    itag=1047
     call mpi_comm_rank(mpi_comm_world,nrank,ier)
     print *,' hello world from task ',nrank
     if(nrank .eq. 1) a=99
     if(nrank .eq. 1)
1      call mpi_send(a,1,mpi_real,5,itag,mpi_comm_world,ier)
     if(nrank .eq. 5)
1    call mpi_recv(a,1,mpi_real,1,itag,mpi_comm_world,STATUS,ier)
     print *,' A is ',a, ' AT TASK ',nrank
     call mpi_finalize(ier)
      stop
        end
```

This program sends a message from task 1 to task 5 and prints from all tasks to show the value of a variable in each task.

```
 program f1
 include 'mpif.h'
 integer STATUS(mpi_status_size)
integer itag
 call mpi_init(ier)
 a=0
itag=1047
 call mpi_comm_rank(mpi_comm_world,nrank,ier)
 print *,' hello world from task ',nrank
 if(nrank .eq. 1) a=99
 if(nrank .eq. 1)
1   call mpi_send(a,1,mpi_real,5,itag,mpi_comm_world,ier)
 if(nrank .eq. 5)
1  call mpi_recv(a,1,mpi_real,1,itag,mpi_comm_world,STATUS,ier)
 print *,' A is ',a, ' AT TASK ',nrank
 call mpi_finalize(ier)
  stop
    end
```

MPI_COMM_WORLD is an MPI *Communicator.*

MPI communcators define groups of tasks. Many MPI codes have just one, the default MPI_COMM_WORLD.It is possible to create others from a subset of your tasks and operate on them only. Coupled atmos-ocean models do this.. In elementary and intermediate codes this is just an extra argument you need to put in most calls.

MPI_REAL defines the type of data you will send/receive.

STATUS is scratch space used by mpi_receives.

mpif.h brings in various MPI defaults and data structures MPI needs
mpi_init creates the comms infastructure
mpi_comm_rank, tells which task you are
mpi_send sends a message
mpi_recv receives a message
mpi_finalize cleans up, flushes buffers and exits tasks.

call mpi_send(a,1,mpi_real,5,itag,mpi_comm_world,ier)

- a is the variable (beginning address) you send.
- 1 is the length (number of words) you send
- mpi_real is the type of data you are sending

- 5 specifies where you are sending to (task 5)
- itag specifies the message number (1:32K). This is used by the receiver to separate this message from others in the queue. It must be unique in the queue (not a problem usually)
- MPI_COMM_WORLD defines a group of tasks (it is defined in mpif.h and is the default for mpi calls. You can make others but for now just treat it as an extra thing you need.

1 call mpi_recv(a,1,mpi_real,1,itag,mpi_comm_world,STATUS,ier)

call mpi_send(a,1,mpi_real,5,itag,mpi_comm_world,ier)

- a is the variable (beginning address) you send.
- 1 is the length (number of words) you send
- mpi_real is the type of data you are sending

- 5 specifies where you are sending to (task 5)
- itag specifies the message number (1:32K). This is used by the receiver to separate this message from others in the queue. It must be unique in the queue (not a problem usually)
- MPI_COMM_WORLD defines a group of tasks (it is defined in mpif.h and is the default for mpi calls. You can make others but for now just treat it as an extra thing you need.

1 call mpi_recv(a,1,mpi_real,1,itag,mpi_comm_world,STATUS,ier)

These arguments match the send's except for STATUS. STATUS is scratch space dimensioned MPI_STATUS_SIZE (specified in mpif.h) used internally by the receive libraries. Just dimension it and put it in the argument list. It's another "extra" but unlike MPI_COMM_WORLD there is no user need to ever work with it.

Note the send sent to task 5. The receive gets from task 1. (fourth argument).

More about point to points

For each point to point comm there are two tasks, the sender and receiver.
There is also the unique tag to identify the messages at the receiving end.
If a sent message is not received or a receive is called for an unsent message, the task hangs. This mistake is very common.
Hangs also occur if the tags don't match.

If the conditional send is removed from the simple program, task 5 will complete when it gets the task 1 message. Most others will hang at the send to 5 which is not looking for these other messages. Task 1 will hang at the receive.

The fundamental comms operation in grid point models is point to point send/receives of fairly large messages.

Grid point models evaluate finite difference operators to define spatial derivatives.

Each task takes a portion of a grid.

Finite difference operators on portion edges are not possible

Problem is solved by replicating neighbor edges.. these are often called HALOs. The replication is done by sending the edge row or column to the neighbor task and in turn receiving that task's edge information. For a grid each task needs a side from each of its four neighbors.

The rest is bookkeeping!

Halos or ghost points

- Consider 16x16 domain
- Each subdomain is 4x4 on 16 processors
- To evaluate finite differences at border points we need the neighbor borders.
- Subdomain arrays are dimensioned 0:5,0:5
- Row zero is populated by neighbor below's row 4
- Row 5 is populated by neighbor above's row 1
- Same is done for columns. Finite differences can then be done from 1:4,1:4 on all subdomains

Blocking v.s nonblocking

The previous sends block. Consider

Task 1	task 5
Call mpi_send(.,.,.,5,.,.,.)	call mpi_send(.,.,.,1,.,.,.)
Call mpi_recv(.,.,.,5,.,.,.,.)	Call mpi_recv(.,.,.,1,.,.,.,.)

The send to 5 will not return till 5 has gotten the message. 5 won't process

The message till the send to 1 returns which won't happen till 1 gets the message.

But 1 can't process the message because it is waiting for 5 to receive.

This is called a DEADLOCK.

Blocking v.s nonblocking

One way to avoid this is to use nonblocking send/receive. The send queues the array for eventual sending and continues or the receive spawns a request to look for a message and get it when it appears. Usually only one side needs to be nonblocking

Task 1	task 5
call mpi_isend(.,.,.,5,.,**IR**,.)	call mpi_isend(.,.,.,1,.,.,**IR**,.)
call mpi_recv(.,.,.,5,.,.,.,.)	call mpi_recv(.,.,.,1,.,.,.,.)

IR here is a request handle. It is set in the MPI libraries and is OUTPUT for the nonblocking send/receives. It is needed to test for completion of the nonblocking operation. You MUST make sure it is complete before doing anything with the variable being sent (overwriting or deallocating it), or received (using, overwriting or deallocating it). This is done with

MPI_WAIT(ir,status,ier) where status is an array of the same size as the status in mpi_recv or with a following blocking call communicating with the same task the nonblocking call did (the latter is common). The above simple code does not need a test but when in doubt include it.

Blocking v.s nonblocking

MPI_SEND is NOT guaranteed to block. Small messages are copied to a buffer. You are guaranteed to be able to use the sent variable or delete it after MPI_SEND returns control. (The MPI_SSEND call is guaranteed to block until the message is sent.)

It is also possible to communicate in the background while computing.

Call mpi_isend(…….)

Compute.. Compute .. .. ..

Call mpi_wait(…)

This is more useful on machines with dedicated comms processors. On ours, the interrupts from MPI use of the cpus are expensive and it's often best to just communicate and then compute after comms are done.

Collectives

- Operations so far have acted on a single pair of tasks. (still fast because the N tasks, do their comms concurrently).
- A second large class of operations is COLLECTIVES. These perform the same operation on all tasks and block on all tasks until the last task starts the operation and satisfies other tasks' internal implementation prereqisites. You don't need to know these prerequisites, the MPI library writers dealt with them. But collectives do NOT block until all are finished however it's usually pretty close. For most purposes you can think of them as blocking.
- Every task in the group has to call the collective or the mpi job hangs at the collective call. **Watch out for collectives in conditional code!**

Collective examples

- MPI_BCAST
- MPI_SCATTER
- MPI_GATHER (MPI_ALLGATHER,MPI_GATHERV)
- MPI_BARRIER
- MPI_ALLTOALL (MPI_ALLTOALLV) (this one is core of GFS comms)
- MPI_REDUCE.

MPI_BARRIER

- MPI_ BARRIER(comm,ier).
- This one is used to sync all tasks. Contrary to common belief it is VERY CHEAP. However any algorithm load imbalance will surface as barrier time for all but the slowest task. Requirement for syncing is what creates the cost . Example!
- Task1(fastest) task5(slowest)
- Call relax(a)(25sec) call relax(a) 75 sec
- Call mpi_barrier(..)50sec call mpi_barrier()1ms
- You've lost 50 seconds of compute time on task 1.

. Often used with timers to DIAGNOSE load imbalance

Collectives

Consider

```
if(ntask .ne. 5) then
      call mpi_ barrier(mpi_comm_world,ier)
endif
```

All of the barrier calls will hang except in task 5 which will run to the next collective or mpi_finalize (and hang there). 5 didn't enter the collective so everyone else stuck waiting for it

MPI_BCAST

- **MPI_ BCAST(array,num,datatype,root,comm,ier)**
- Array is the data to replicate on all tasks,
- num is the length (number of words) to send, datatype specifies real, integer, character, ..,..,
- Root is the task number that initially has the data
- Comm is the communicator name (MPI_COMM_WORLD in this class)
- Ier is an output error status.
- This broadcasts "Array" initially on task "root" to all tasks.
- Don't use a send loop to do this. Broadcast cost is log(N) while sendloop cost is N. I have found send loops even in well worked over benchmarks!

MPI_SCATTER

- This takes an array and scatters it in chunks to the various other tasks.
- **MPI_SCATTER(full_array,len,datatype,task_array,len,datatype,root,comm,ier)**
- One common use is reading a full domain on one task and sending the subdomains each task will handle to the various tasks.
- MPI_SCATTERV supports chunks of varying sizes.
- **MPI_SCATTERV(full_array,len_array,disp_array,datatype, task_array,len,datatype,root,comm,ier)**
- disp_array contains displacements in full_array. The displacement for task N defines how many words after full_array(1) to begin the send, and len_array(n) defined the number of words to send.

MPI_GATHER

- **MPI_ GATHER(array,num,datatype,arrayall,num,datatype,root,comm,ier)**
- **Array** is the remote array on the tasks
- **num** is the length (number of words) to receive,
- **datatype** specifies real, integer, character, ..,..,
- **Arrayall** is the array to contain all data from all tasks. **Num** and **datatype** are duplicated in next two args.
- **Root** is the task number that is gathering all of the data (perhaps to write to a file)
- **Comm** is the communicator name (MPI_COMM_WORLD in this class)
- Ier is an output error status.
- This gathers "Array" from all tasks to a much larger (num*ntasks) master array.
- MPI_ALLGATHER is similar but populates arrayall on ALL tasks.
- MPI_REDUCE is sometimes used on a problem size data structure with all nontask parts zeroed out. This indeed generates the entire domain but is more expensive than a gather.

MPI_ALLTOALL

- **MPI_ALLTOALL(dataout,len,datatype, datain,len,datatype,comm,ier)**
- This sends slices of dataout to all tasks and gathers slices of datain from all tasks. This is useful in full domain transposes and is best illustrated by example.

COLLECTIVES

- MPI_ALLTOALL(dataout,len,datatype,Datain,len,datatype,comm,ier)
- Consider a 64 level domain decomposed by level on 64 processors {T(256,512,64)}
- There are 256 lats and 512 lons. Each task does one level This is ideal for parallel FFT
- dataout maps to T(256,512,L)
- You need vertical communication to do your physics. Change decomposition to lon,lev.
- datain maps to T(256,lon:lon+7,64).
- Task 1 sends 512,1:8 to task 1, 512,9:16 to task 2 .. ..
- Task 1 receives 512,1:8,1 from 1 then 512,1:8,2 from 2.
- Task 1 is sending all 512 lons of level 1 in 8 lon slices to the 64 tasks.
- Task 1 is getting lon 1-8 from all levels from the 64 tasks
- After physics, this transpose is inverted to go back to level decomposition.
- This is the core of spectral model comms although there each task does some lats and some lons for load balance. That requires building a buffer before alltoall is called.. Newer GFS decomposes by wavenumber but the dominant communication is still an alltoall to do a transpose

COLLECTIVES

- There is an MPI_ALLTOALLV similar call that moves different amounts of data to each processor and receives different amounts from each processor. MPI_GATHERV also gathers data with tasknumber dependent lengths.

COLLECTIVES MPI_REDUCE

- MPI_REDUCE does summations and other reductions across tasks.

MPI_REDUCE(array,arraysum,len,datatype,**op**,root,comm,ier)

OP is an input integer mapping to an operation

THESE ARE DEFINED IN mpif.h or you can make your own.

MPI_REDUCE_SCATTER scatters arraysum to the different tasks. It is analagous to calling MPI_SCATTER on arraysum.

REDUCTION OPERATORS

- MPI_MAX max
- MPI_MIN min
- MPI_SUM sums
- MPI_PROD multiplies
- MPI_LAND,MPI_LOR,MPI_BOR,MPI_LXOR, MPI_BXOR logical operations.

I/O

- Each task can handle files independently. Stdin, stdout and stderr are special cases which are handled as a single stream by all tasks.
- Often one task handles the I/O and communicates with the others to get/send the relevant data.
- Read, scatter, integrate, gather, write is common scenario.
- Some codes write their subdomains to task specific files. Collection is done later offline.
- I/O often blocks scaling beyond ~100 tasks until addressed and optimized.. Then not a big issue.
- Two primitive optimizations are to write all subdomains in parallel and to send all data to be written, to a special task at switch speeds and then have that task write to disk or network in the background. Both NAM and GFS do the latter.
- MPI-2 standard includes the concept of MPI_IO, collective operations on data distributed across tasks. This is the most likely part of MPI_2 to be implemented at a transitioning site but it's beyond scope of this talk.

UTILITY CALLS

- MPI_COMM_RANK(comm,nrank,ier) tells which task you are
- MPI_COMM_SIZE(comm,ntasks,ier) tells how many tasks there are.
- MPI_GET_PROCESSOR_NAME(char,len,ier) returns the machine name for each task in a character variable of length len.

COMPILING AND RUNNING

- On the IBM machines the MPI compilers are
- Mpxlf,mpxlf90,mpxlf95,mpxlf_r,mpxlf90_r,mpxlf95_r. On blue/white we also have these with the "ncep" prefix e.g ncepmpxlf.
- These bring in the needed includes or modules and libraries. They are wrappers around the fortran compiler and loader with the many additional options needed to build MPI programs hidden from the users.
- mpxlf f.f is all you need to do. On other systems consult local documentation

COMPILING AND RUNNING

- To run a small number (say 4) of MPI tasks interactively
- 1. export MPI_PROCS=4; export MP_RMPOOL=1
- 2. ./a.out <input >output 2>err
- MPI collects the prints from various tasks and streams them all to stdout and stderr. MPI also reads stdin and broadcasts to all tasks so you don't need to worry about that little detail.

RUNNING

- For larger MPI jobs you need LoadLeveler, the IBM batch node scheduler.
- One typically asks for a number of tasks on a smaller number of nodes, specifies stdout location and job class and submits the job.
- Consult ibmdocs.ncep.noaa.gov for LoadLeveler documentation but a few examples will be presented here.

RUNNING

LL job (minimal)

```
#!/bin/ksh
#@network.MPI=csss,shared,us
#@ job_type=parallel
#@ class=dev        ! Classes are 1 and dev for most users.  Dev uses most of the machine. I is interactive
#@ total_tasks=20
#@ wall_clock_limit = 01:02:09
#@ node=10
#@ queue
 {setup}  <--- this is the commands necessary to set up inputs for a.out
./a.out
exit
```

RUNNING

All of your LL job command except for explicitly parallel ones run SERIALLY on one node. In the job above only your a.out will run on all of the nodes.

To submit the job do

llsubmit $file where $file contains the directives and script to be run. It will be queued for scheduling.

The llsubmit path is /usr/lpp/LoadL/full/bin

RUNNING

LL job (minimal)

```
#!/bin/ksh
#@ output=o    specifiec location of stdout
#@ error=e   specifies job stderr location (here e)( I like short names)
#@ job_type=parallel
#@network.MPI=csss,shared,us
#@ class=dev   (dev is for big jobs, 1 is for small and interactive)
#@ total_tasks=20
#@ wall_clock_limit = 01:02:09
##@ task_geometry=( (1,9)(3,11)(2,4) .. ..    Rearranges task layout on processors
#@ node=10
##@ blocking=unlimited                  allocates tasks wherever there are slots
#@ queue
 (setup)   <--- this is the commands necessary to set up inputs for a.out
./a.out
exit
```

VERY USEFUL ENVIRONMENT VARIABLES

- Set MP_LABELIO=yes. This prefixes task number to each line of stdout and stderr so you can see which task printed it.
- Set MP_RETRYCOUNT to 30 or so and MP_RETRY=25 or so. Interactive jobs will then try every 25 seconds for up to 30 tries to get nodes. (default is 0 for retrycount so if interactive nodes aren't available, you just error out). (irrelevant in LL jobs.)
- For interactive jobs MP_RMPOOL must be set to 1.
- MP_EAGER_LIMIT controls buffering and is useful for debugging (0 no buffering is of special interest)

TIMING COMMS

- To find out how long a call is taking insert a time call before and after (low level)

```
T1=rtc()
Real(kind=8)rtc,t1,t2
Call mpi_all2all()
T2=rtc()
Call_time=(t2-t1) .. This is in seconds.
```

TIMING COMMS

You can get a summary of MPI activity and time consumed by linking with

-lmpitrace in your load parameters e.g

Mpxlf your.code –lmpitrace

This produces N reports in mpi_profile.tasknum, one per task. These

Are in your current working directory.

SCALING

MPI_BCAST logN

Sendloop to broadcast (N)

MPI_SCATTER constant

MPI_GATHER constant
MPI_ALLGATHER,MPI_GATHERV)2constant

MPI_BARRIER epsilon*n

MPI_ALLTOALL (MPI_ALLTOALLV) (this one is core of GFS comms) 1/n

MPI_REDUCE. 1/nlogn (??)

- 1/n implies perfect scalability constant implies no scalability, N implies time ***grows*** linearly with taskcount

MPI_WRAPPERS

- These are a too little known part of the standard.
- Originally developed to support tracing and profiling libraries (example PALLAS Vampir and our mpitrace libraries).
- Also useful for debugging.
- You write a subroutine with the same binding as an MPI routine and give it the same name (e.g mpi_send).
- This routine after setting a timer or checking arguments, calls PMPI_SEND with identical binding.
- Each MPI routine has a second entry name with a P prefix.

Debugging

- Hangs??

Use prints to determine hang point.

Check for point to point send/receive mismatch.

This could be lack of the call at all, a tag mismatch or a processor number mismatch.

Check for conditional code in barriers.

(usually the one that DOESN't call the collective progresses further and hangs at the next collective.

Debugging

- Hangs??
- Sometimes hangs occur well after an incorrect send/receive pair if buffering is done.
- MPI Standard does not address buffering schemes
- MP_EAGER_LIMIT controls buffering. Set to 0 to disable buffering (but then expect a slowdown)

Hangs then occur at the trouble point.

Debugging

- Segmentation faults
- Usually indicate the array you are receiving is too short and the incoming data stream is overwriting other stuff.
- Also happens from argument mismatches and it can be subtle if the datatypes don't match.
- Also happens in nonblocking calls when the array to be sent/received is deallocated before completion. Exiting out of the routine that made the call before completion can cause this.
- Sometimes happens deep in the MPI libraries from load imbalance. Inform sysadmins of these and also reduce the load imbalance

Debugging

- "Terminated" messages. Usually means either the job ran out of time or another task exited with nonzero return codel ALL tasks are sent a signal 15 (kill gently) if any one exits with nonzero return code.
- Don't use fortran STOP nn where nn is a nonzero number.
- You must call MPI_FINALIZE from all tasks otherwise your I/O buffers won't flush and your output files may be incomplete. Call MPI_ABORT in place of a stop and don't call MPI_FINALIZE in conditional code.
- (call mpi_abort(mpi_comm_world,ireturn,ier)

Debugging

- Send inquiries to ncep.list.sp-support@noaa.gov or use ticket system in http://ibmdocs/hd.
- App person is george.vandenberghe@noaa.gov 301-763-8115x7119

Performance considerations

Don't oversubscribe the cpus (running more tasks than cpus on a node or product of taskcount*threads/task >cpus on a node.

Try to pass data as a few large messages rather than many small ones

If you use threads, run with

#@node_usage = not_shared

This is especially important if you run a lot of tasks.

Performance considerations

- Be careful to make sure each task has about the same amount of work. If not Load Imbalance will waste cpus.
- This is admittedly not always easy (convective hot spots in physics are one example)
- VERY large numbers of tasks may benefit from running n-1 tasks per node rather than N where N is cpu count on the node.
- Running fewer tasks gets faster turnaround and is more efficient (if you have the wall time)

Examples

- An example of a reduce is in /nfsuser/g02/courses/mpi/reduce.f
- An example of a scatter, transpose, gather is in /nfsuser/g02/courses/mpi/coll.f
- Will work on a simple halo exchange but not ready for 6/20. This will be in laplace.f
- A sample LL job is in /nfsuser/g02/courses/mpi/j